Code Generation for Embedded Processors

Embedded Systems

10.07.2007

Daniel Kästner
AbsInt Angewandte Informatik GmbH
kaestner@absint.com

What's it all about?

C code (FIR Filter):

```
int i,j,sum;
for (i=0;i<N-M;i++) {
  sum=0;
  for (j=0;j<M;j++) {
    sum+=array1[i+j]*coeff[j];
  }
  output[i]=sum>>15;
}
```

Compiler-generated code (gcc):

```
.L21: add %d15,%d3,%d1
      addsc.a %a15,%a4,%d15,1
      addsc.a %a2,%a5,%d1,1
      mov %d4,49
      ld.h %d0,[%a15]0
      ld.h %d15,[%a2]0
      madd %d2,%d2,%d0,%d15
      add %d1,%d1,1
      jge %d4,%d1,.L21
```

Hand-written code (inner loop):

```
_8: ld16.w d5, [a2+]
    ld16.w d4, [a3+]
    madd.h e8,e8,d5,d4ul,#0
    loop a7,_8
```

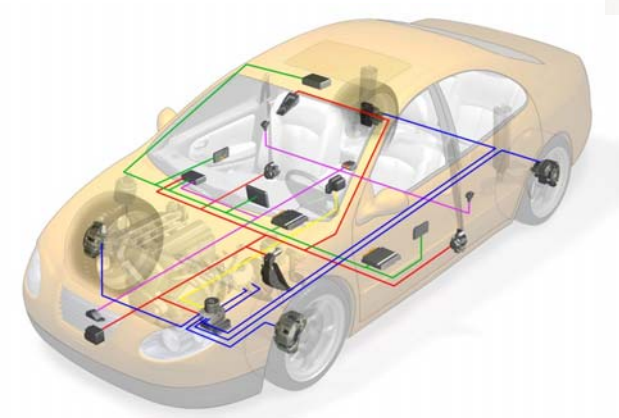
- ✓ **6 times faster!**
- ✓ Processor performance depends on quality of code generation.
- ✓ Good code generation is a challenge.

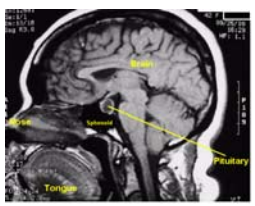
Overview

- A Classification of microprocessors
- Characteristics of embedded processors
- Code generation:
  - Program representations
  - Basic algorithms
  - Phase Coupling Problems
- Advanced Topics

Embedded Processors: Application Areas

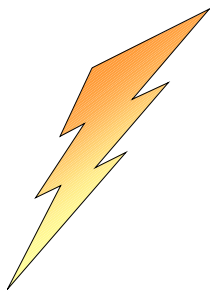
- Automotive
- Avionics
- Telecommunication
- Consumer electronics
- Healthcare technology





Design of Embedded Systems: Goal Conflict

- High performance requirements
- Increasing software complexity
- High dependability



- Low cost
- Low power consumption
- Short product and development cycles

Types of Microprocessors

- Complex Instruction Set Computer (CISC)
  - large number of complex addressing modes
  - many versions of instructions for different operands
  - different execution times for instructions
  - few processor registers
  - microprogrammed control logic
- Reduced Instruction Set Computer (RISC)
  - one instruction per clock cycle
  - memory accesses by dedicated load/store instructions
  - few addressing modes
  - hard-wired control logic

Example Instruction (IA-32)

Opcode	Instruction	Description
04 *ib*	ADD AL,*imm8*	Add *imm8* to AL
05 *iw*	ADD AX,*imm16*	Add *imm16* to AX
05 *id*	ADD EAX,*imm32*	Add *imm32* to EAX
80 /0 *ib*	ADD *r/m8,imm8*	Add *imm8* to *r/m8*
81 /0 *iw*	ADD *r/m16,imm16*	Add *imm16* to *r/m16*
81 /0 *id*	ADD *r/m32,imm32*	Add *imm32* to *r/m32*
83 /0 *ib*	ADD *r/m16,imm8*	Add sign-extended *imm8* to *r/m16*
83 /0 *ib*	ADD *r/m32,imm8*	Add sign-extended *imm8* to *r/m32*
00 /r	ADD *r/m8,r8*	Add *r8* to *r/m8*
01 /r	ADD *r/m16,r16*	Add *r16* to *r/m16*
01 /r	ADD *r/m32,r32*	Add r32 to *r/m32*
02 /r	ADD *r8,r/m8*	Add *r/m8* to *r8*
03 /r	ADD *r16,r/m16*	Add *r/m16* to *r16*
03 /r	ADD *r32,r/m32*	Add *r/m32* to *r32*

Execution time:
- 1 cycle: ADD EAX, EBX
- 2 cycles: ADD EAX, memvar32
- 3 cycles: ADD memvar32, EAX
- 4 cycles: ADD memvar16, AX

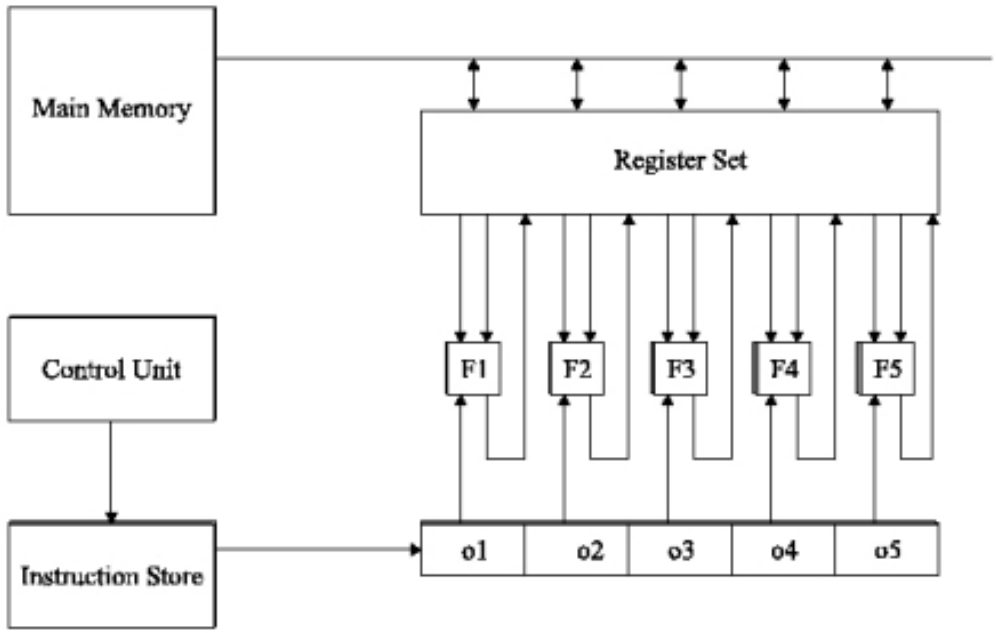
Instruction Width:
between 1 byte (NOP) and 16 bytes

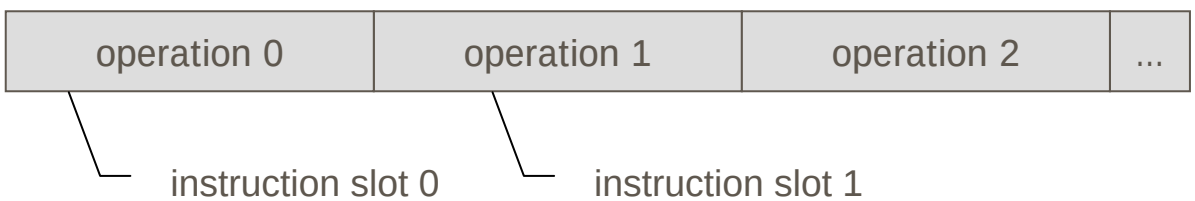
Types of Microprocessors

- Superscalar Processors
  - subclass of RISCs or CISCs
  - multiple instruction pipelines for overlapping execution of instructions
  - parallelism not necessarily exposed to the compiler

- Very Long Instruction Word (VLIW)
  - statically determined instruction-level parallelism (under compiler control)
  - instructions are composed of different machine operations whose execution is started in parallel
  - many parallel functional units
  - large register sets

VLIW Architectures





Architectural Valuation

- More efficient architectures will use less energy to complete the same task on the same generation CMOS solid state technology
- Power consumption $P = CV^2 f\Delta N$
  - C : Capacitance
  - V : CPU Core Voltage
  - f : CPU clock frequency
  - DN : Number of gates changing state

- Architectural specialization as a measure for how well the architecture fits a given target application.
- Estimation of architectural specialization: Performance per power.

Architectural Valuation

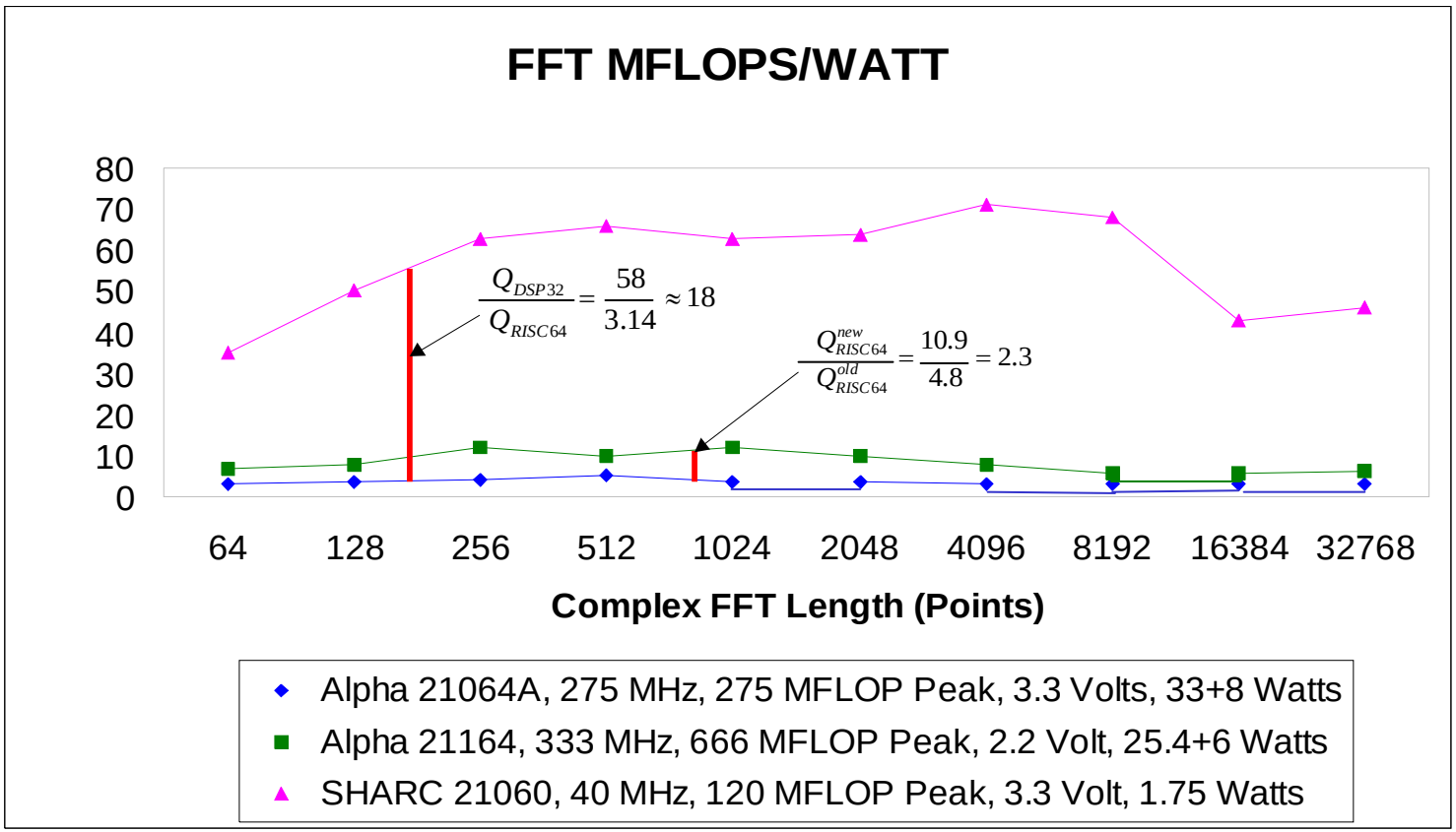
- Observations:
  - Higher performance by increasing the clock frequency does not change the performance per power ratio

$$\frac{Performance}{Power} = \frac{\frac{O}{Cycle} * \frac{Cycle}{Time}}{CV^2 f \Delta N} = \frac{Of}{CV^2 f \Delta N} = \frac{O}{CV^2 \Delta N}$$

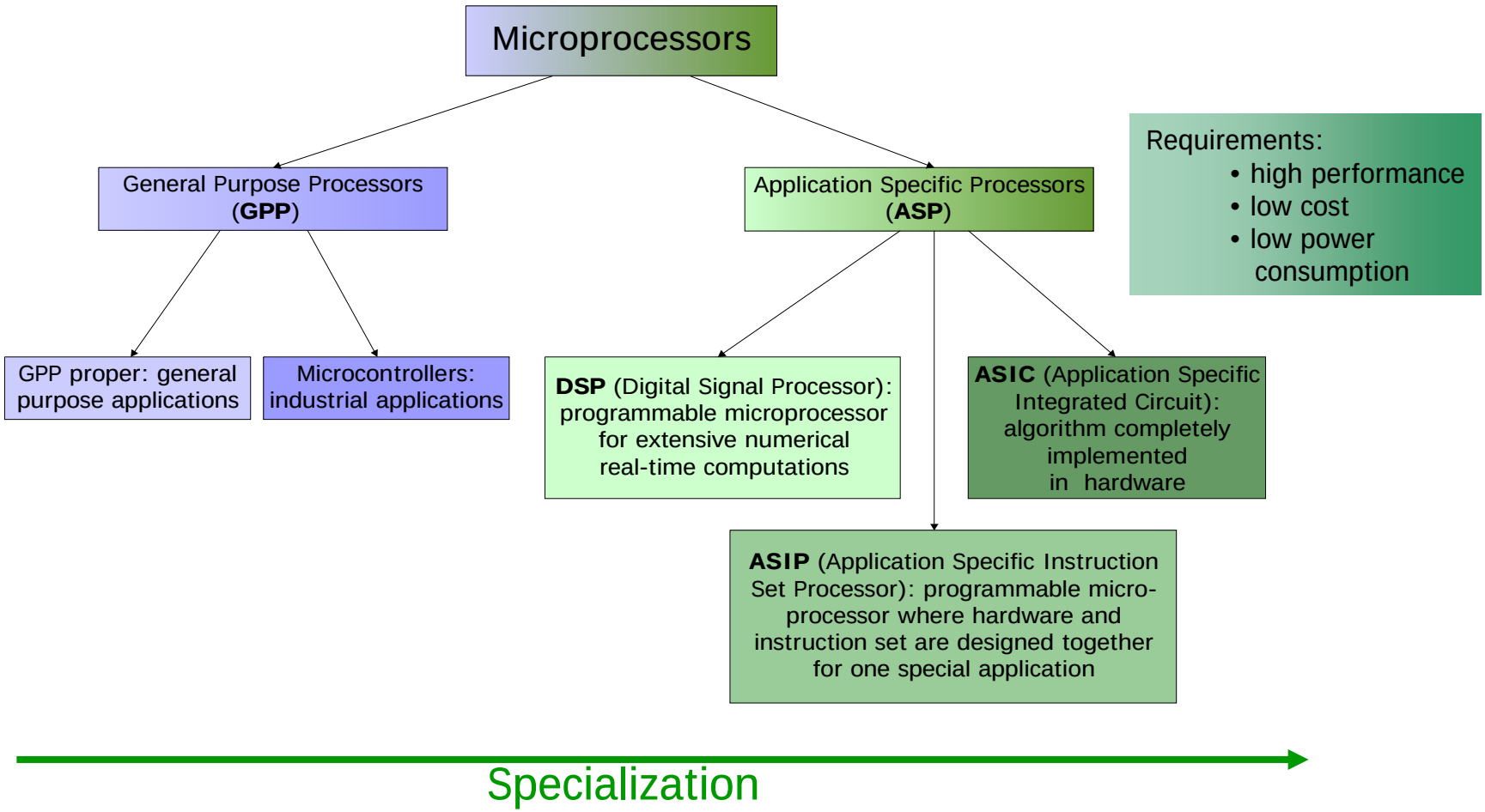
  - A voltage decrease improves performance per power non-linearly

$$\frac{Performance}{Power} = \frac{O}{CV^2 f \Delta N}$$

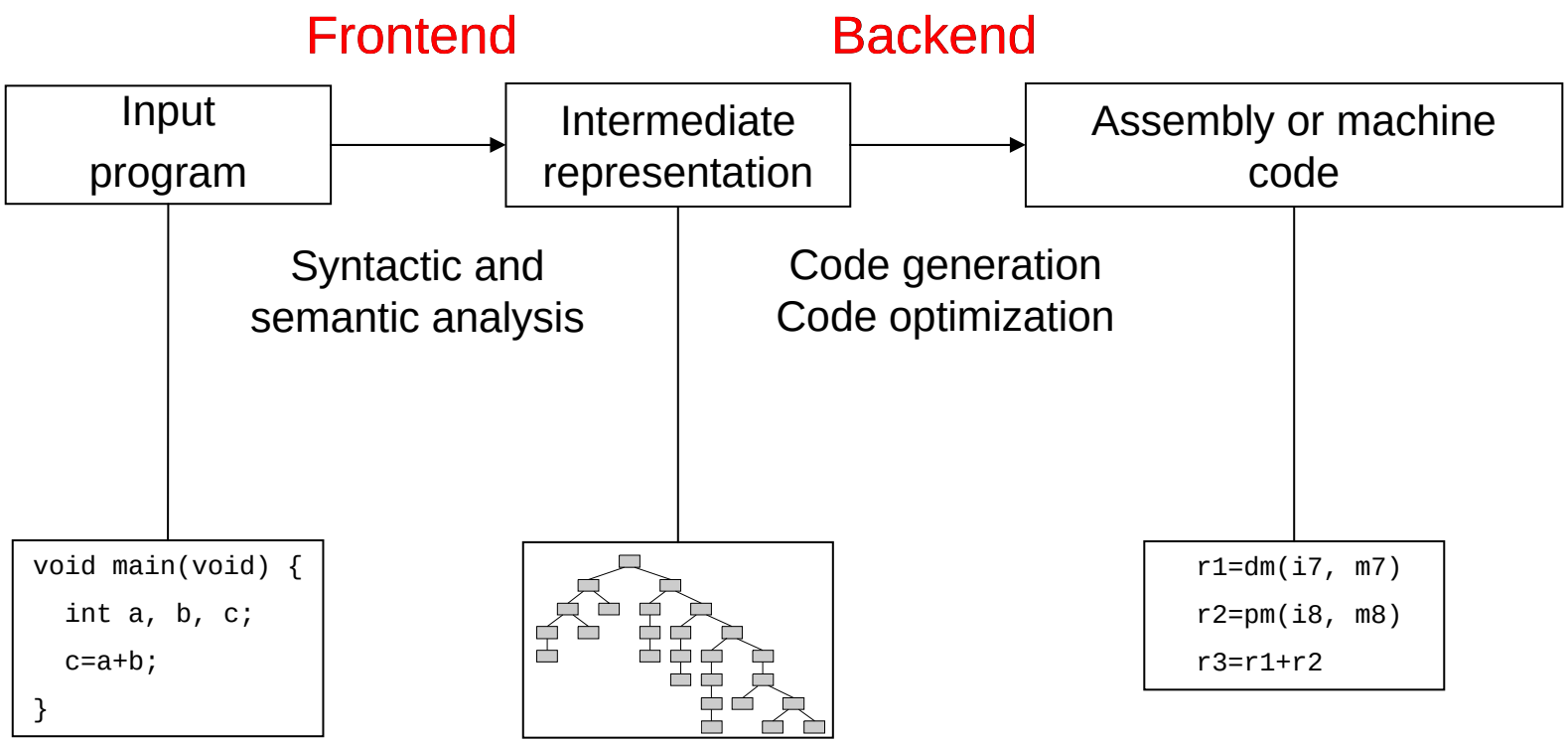
Comparison of Performance Per Power Ratios



Classification of Microprocessors

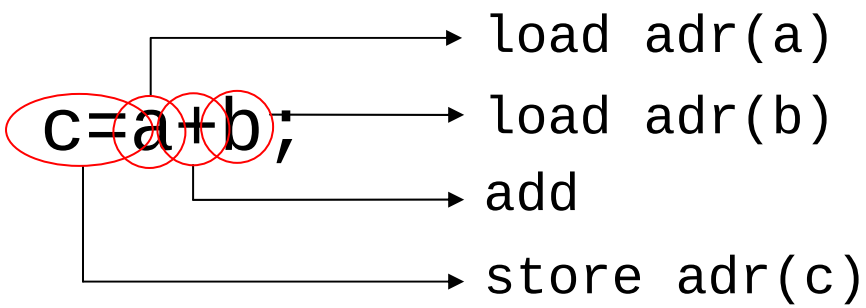


Compiler Structure

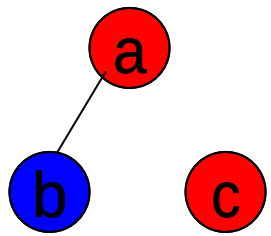


Backend: Main Phases

Code selection



Register allocation



Instruction scheduling

```
r1=load adr(a) || r2=load adr(b)
r1=add r1, r2
store adr(c), r1
```

Main Tasks of Code Generation (1)

- Code selection: Map the intermediate representation to a semantically equivalent sequence of machine operations that is as efficient as possible.

- Register allocation: Map the values of the intermediate representation to physical registers in order to minimize the number of memory references during program execution.
  - Register allocation proper: Decide which variables and expressions of the IR are mapped to registers and which ones are kept in memory.
  - Register assignment: Determine the physical registers that are used to store the values that have been previously selected to reside in registers.

Main Tasks of Code Generation (2)

- Instruction scheduling:
  Reorder the produced operation stream in order to minimize pipeline stalls and exploit the available instruction-level parallelism.

- Resource allocation / functional unit binding:
  Bind operations to machine resources, e.g. functional units or buses.

The DSPStone Study

- Evaluation of the performance of DSP compilers and joint compiler/processor systems [1994]. Evaluated compilers:
  - Analog Devices ADSP2101,
  - AT&T DSP1610,
  - Motorola DSP56001,
  - NEC mPD77016,
  - TI TMS320C51.
- Hand-crafted assembly code is compared to the compiler-generated code.
- Result: overhead between 100% and 1000% of compiler-generated code is typical !

Characteristics of DSPs

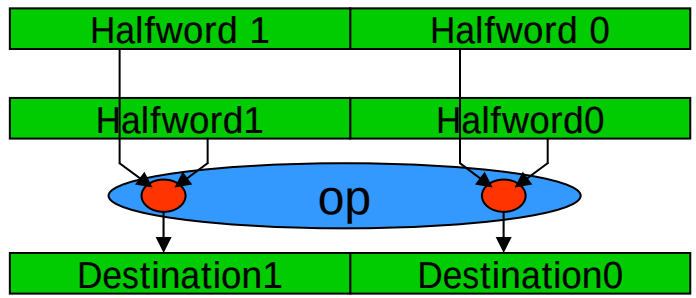
- Multiply-accumulate units: multiplication and accumulation in a single clock cycle (vector products, digital filters, correlation, fourier transforms, etc)
- Multiple-access memory architectures for high bandwidth between processor and memory
  - Goal: throughput of one operation per clock cycle.
  - Required: several memory accesses per clock cycle.
  - Separate data and program memory space: harvard architecture.
  - Multiple memory banks
  - Arithmetic operations in parallel to memory accesses. But often irregular restrictions.
- Specialized addressing modes, e.g. bit-reverse addressing or auto-modify addressing.

Characteristics of DSPs

- Predicated/guarded execution: instruction execution depends on the value of explicitly specified bit values or registers.
- Hardware loops / zero overhead loops: no explicit loop counter increment/decrement, no loop condition check, no branch back to top of loop
- Restricted interconnectivity between registers and functional units -> phase coupling problems.
- Strongly encoded instruction formats: a throughput of one instruction per clock cycle requires one instruction to be fetched per cycle. Thus each instruction has to fit in one memory word => reduction of bit width of the instruction.

Characteristics of DSPs

- SIMD instructions:
  - Focus on multimedia applications
  - SIMD: Single Instruction Multiple Data
  - SIMD-Instruction: instruction operating concurrently on data that are packed in a single register or memory location.




```
a=b+c*z[i+0]
d=e+f*z[i+1]
r=s+t*z[i+2]
w=x+y*z[i+3]
```

$\Rightarrow$

$$\begin{bmatrix} a \\ d \\ r \\ w \end{bmatrix} = \begin{bmatrix} b \\ e \\ s \\ x \end{bmatrix} +_{SIMD} \begin{bmatrix} c \\ f \\ t \\ y \end{bmatrix} *_{SIMD} \begin{bmatrix} z[i+0] \\ z[i+1] \\ z[i+2] \\ z[i+3] \end{bmatrix}$$

Case Study: Infineon TriCore [2002]

C code (FIR Filter):

```
int i,j,sum;
for (i=0;i<N-M;i++) {
  sum=0;
  for (j=0;j<M;j++) {
    sum+=array1[i+j]*coeff[j];
  }
  output[i]=sum>>15;
}
```

Compiler-generated code (gcc):

```
.L21: add %d15,%d3,%d1
      addsc.a %a15,%a4,%d15,1
      addsc.a %a2,%a5,%d1,1
      mov %d4,49
      ld.h %d0,[%a15]0
      ld.h %d15,[%a2]0
      madd %d2,%d2,%d0,%d15
      add %d1,%d1,1
      jge %d4,%d1,.L21
```

Hand-written code (inner loop):

```
_8: ld16.w d5, [a2+]
    ld16.w d4, [a3+]
    madd.h e8,e8,d5,d4ul,#0
    loop a7,_8
```

- ✓ Zero-Overhead Loops
- ✓ SIMD Instructions
- ✓ Auto-modify addressing
- → **6 times faster!**
  (execution in SRAM)

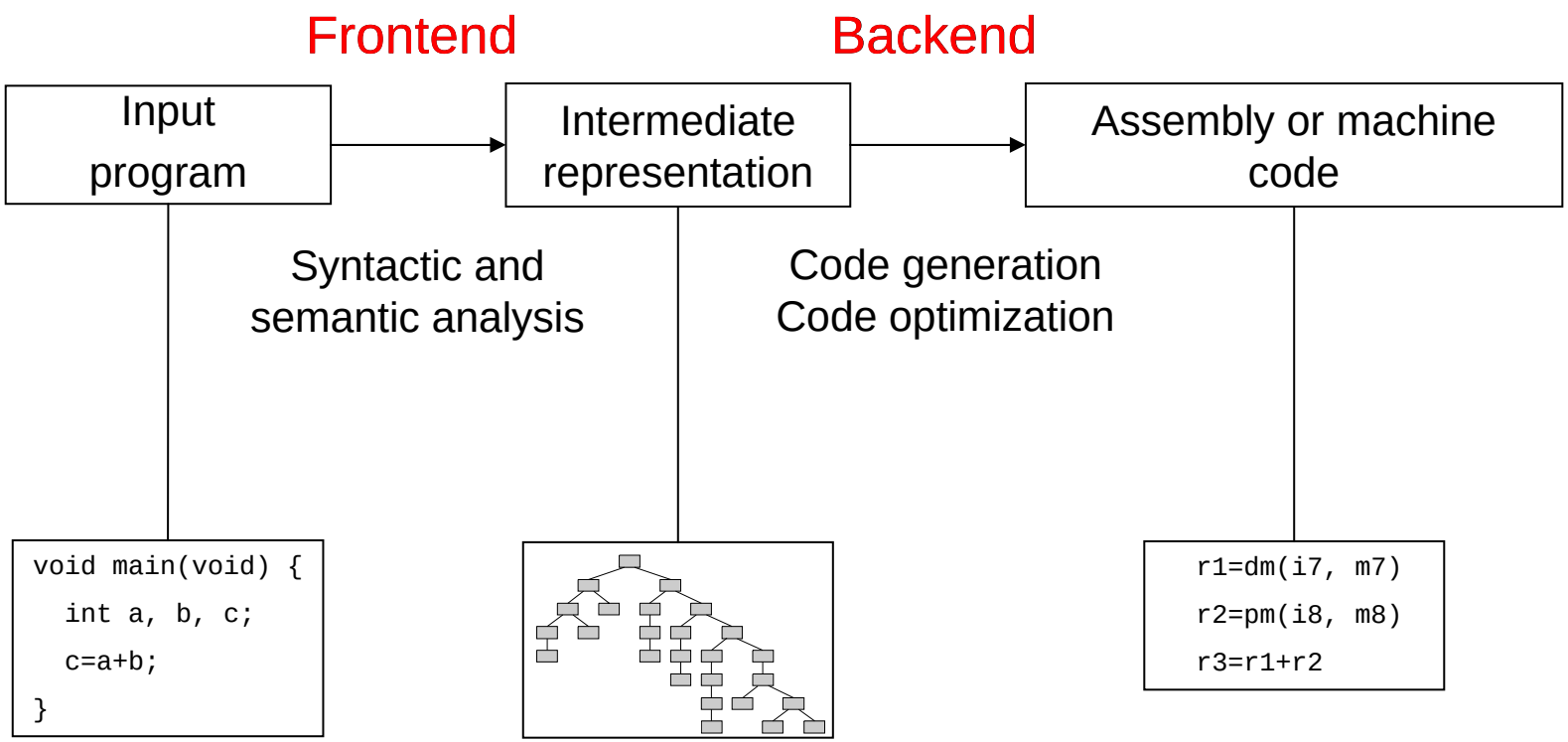
Characteristics of DSPs

- Cost constraints, low power requirements and specialization:
  - Irregularity
  - Phase coupling problems during code generation
  - Need for specialized algorithms

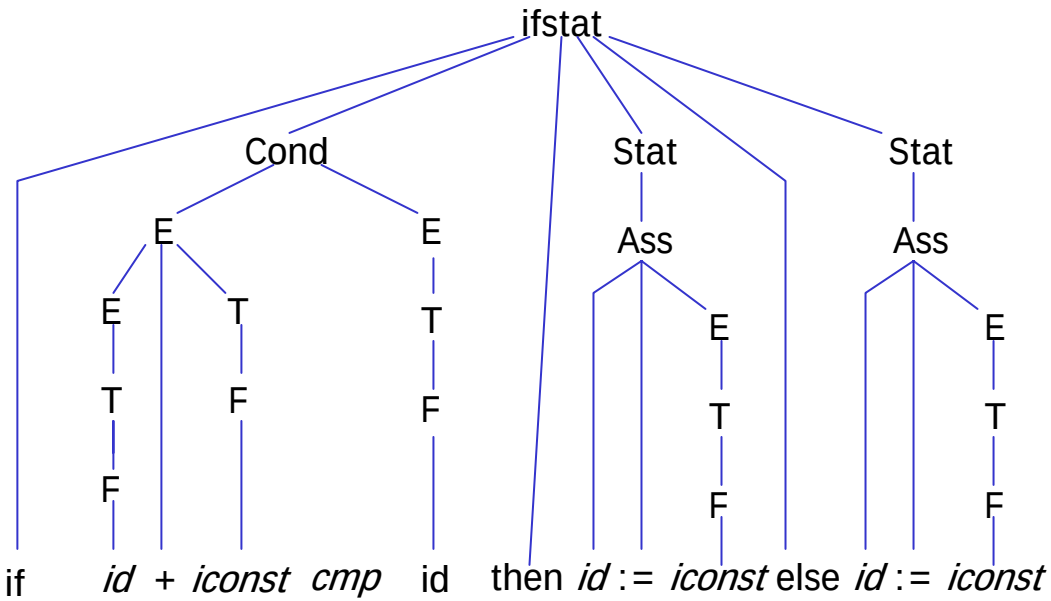
Overview

- ✓ A Classification of microprocessors
- ✓ Characteristics of embedded processors
- Code generation:
  - Program representations
  - Basic algorithms
  - Phase Coupling Problems
- Advanced Topics

Compiler Structure



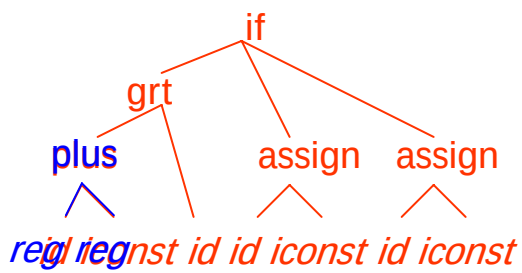
Decorated Abstract Syntax Tree



concrete syntax tree

IR for code selection:

abstract syntax tree



add:

Control Flow Graph (high-level view)

$[y := x]^1$;

$[z := 1]^2$;

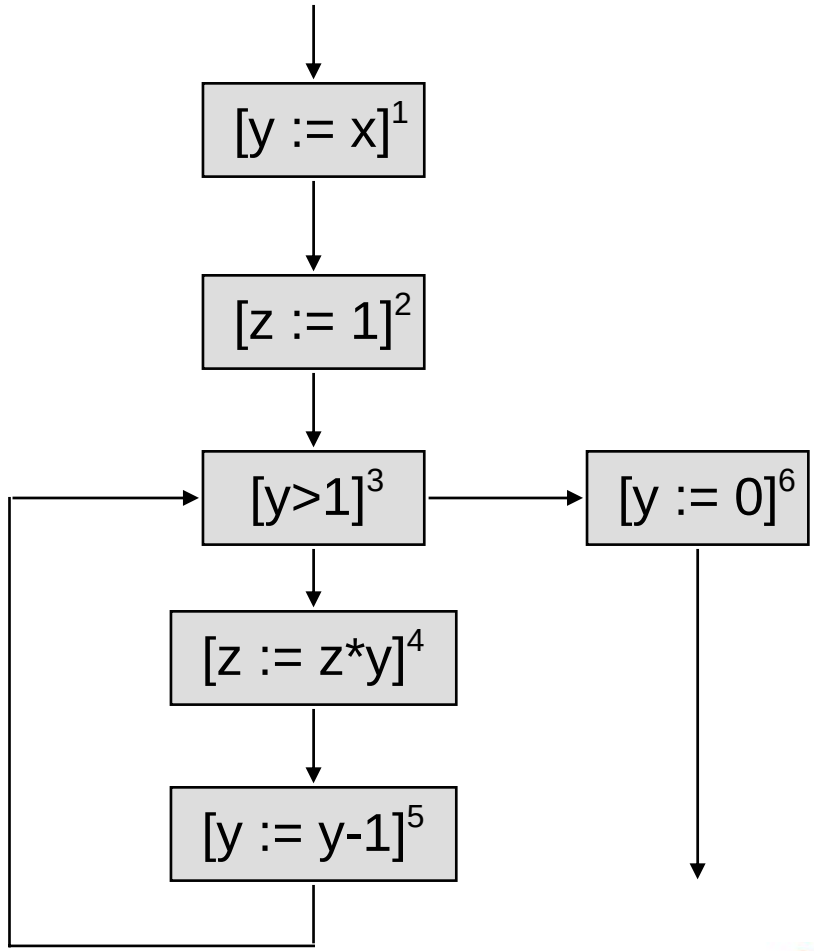
<u>while</u> $[y>1]^3$

<u>do</u> { $[z := z*y]^4$;

$[y := y-1]^5$

};

$[y := 0]^6$



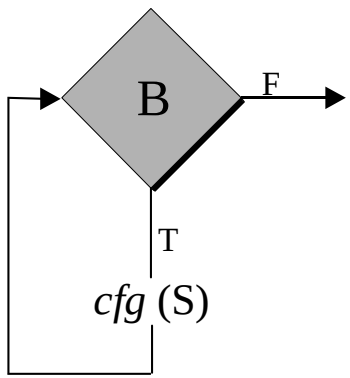
Control Flow Graph

- The control flow graph of a procedure is a directed graph $G_C=(N_C,E_C,n_A,n_\Omega)$ with node and edge labels. For each instruction i of the procedure there is a node n_i that is marked by i. The edges (n,m,λ) denote the control flow of the procedure: $\lambda \in \{T,F,\varepsilon\}$ is the edge label. The nodes for composed statements are shown on the next slide. Edges belonging to unconditional branches lead from the node of the branch to the branch destination. The node n_A is the uniquely determined entry point in the procedure; it belongs to the first instruction to be executed. n_Ω denotes the end node that is reached by any path through the control flow graph.

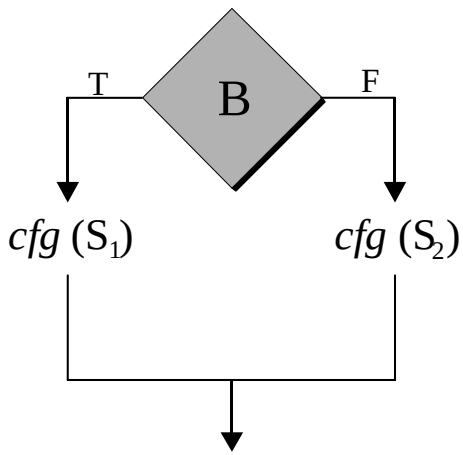
- A basic block in a control flow graph is a path of maximal length which has no joins except at the beginning and no forks except possibly at the end.

Control Flow Graph – Composed Statements

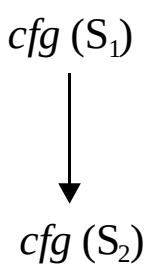
cfg (while B do S od) =



cfg (if B then S_1 else S_2 fi) =

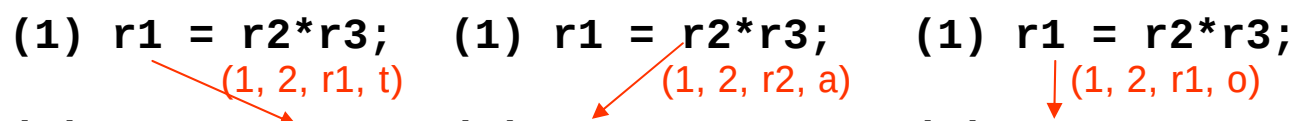


cfg (S_1;S_2) =



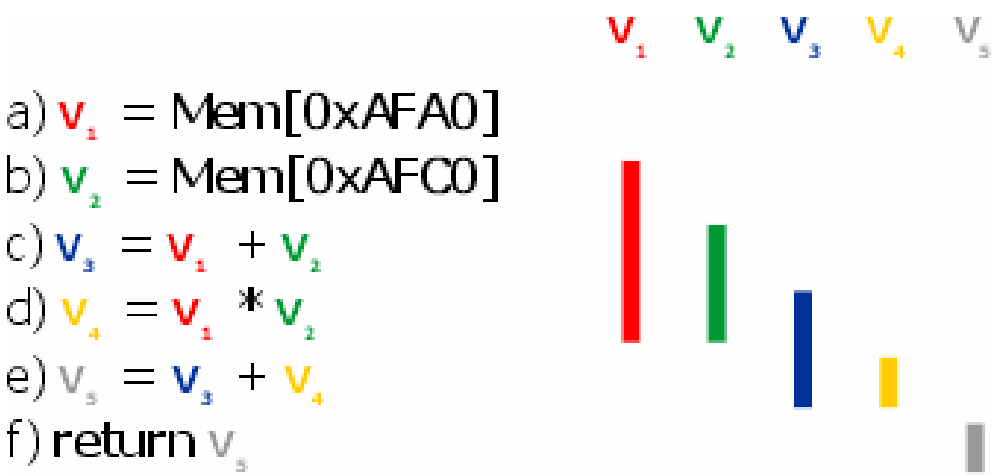
Data Dependence Graph

- Let G_C be a control flow graph. It data dependence graph is a directed graph $G_D=(N_D,E_D)$ with node and edge labels whose nodes are labeled by the operations of the procedure. An edge runs from the node of an operation *i* to the node of an operation *j*, if *i* has to be executed before *j*, i.e. if there is a path from *i* to *j* in the control flow graph and if
  - *i* defines a resource *r*, *j* uses it and the path from *i* to *j* does not contain other definitions of *r* (true dependence, RAW): $(i,j,r,t) \in E_D$
  - *i* uses a resource, *j* defines it and the path from *i* to *j* does not contain any definitions of *r* (anti dependence, WAR): $(i,j,r,a) \in E_D$
  - *i* and *j* define the same resource and the path from *i* to *j* does not contain any uses nor definitions of *r* (output dependence, WAW): $(i,j,r,o) \in E_D$.



Life Range and Register Interference

- A symbolic register (a variable) r is live at a program point *p*, if there is a program path from the entry node of the procedure to *p* that contains a definition of *r* and there is a path from *p* to a use of *r* on which *r* is not defined. The life range of a symbolic register *r* is the set of program points at which *r* is live.
- Two life ranges of symbolic registers interfere, if one of them is defined during the life range of the other. The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.



Register Allocation by Graph Coloring

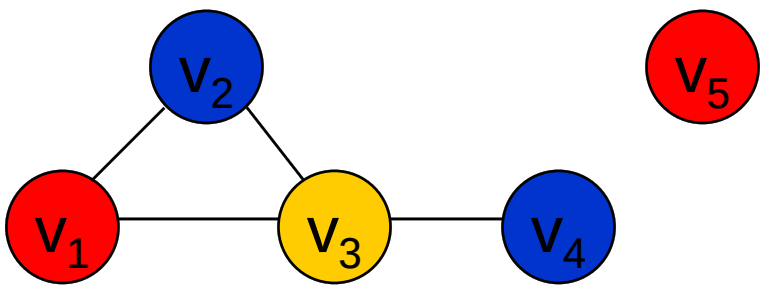
- If *k* physical registers are available, the k-coloring problem must be solved on the register interference graph.
- NP-complete for k>2 -> Use heuristics
- Algorithm:
  - If G contains a node n with degree < k:
    - n and its neighbors can be colored with different colors
    - Remove n from G, decreasing the size of G.
    - G is k-colorable, if we arrive at the empty graph.
  - If G is not empty and there exists no node with degree <k:
    - use heuristics to select one node to remove (spilling)
    - modify program inserting spills at definitions and loads at uses
    - reflect changes in graph.

Heuristics for Node Removal

- Degree of the node: high degree causes many deletions of edges.
- Costs of spilling.

Example: Graph Coloring

```
v1=Mem[0xafa0]
v2=Mem[0xafc0]
v3=v1+v2
v4=v1*v2
v5=v3+v4
return v5
```



Instruction Scheduling

- Definition: Reordering an operation sequence in order to exploit instruction-level parallelism and to minimize pipeline stalls.
  - Complexity: NP-complete.

- Terminology:
  - An operation is a basic machine operation: add, sub, ...
  - An instruction is a set of machine operations that are issued simultaneously (cf. VLIW).
    Example:

```
r3=r1+r4, r3=r10+r14, r11=dm(i6,m6), r12=pm(i15,m15);
```

Instruction Scheduling

- Scope of instruction scheduling:
  - local acyclic instruction scheduling: reordering operations inside basic blocks.
    Standard technique: list scheduling.
  - global acyclic instruction scheduling: reordering operations across basic block boundaries but not across loop boundaries.
    Standard technique: trace scheduling.
  - cyclic instruction scheduling: reordering operations across loop boundaries.
    Standard technique: software pipelining.

List Scheduling

```
SET data_ready;
int cycle=0;

Insert operations without predecessors in the data
   dependence graph into the data_ready set.

while (data_ready ≠ ∅) do {
   cycle = cycle+1;

   Choose operations from data_ready in priority order and
        insert them into the current cycle, until data_ready
   is empty or the insertion leads to a resource conflict.

   Insert all operations into data_ready that can be
   scheduled  in the next cycle without violating data
   dependences.
}
```

List Scheduling

- The priority in which operations from the data ready set are chosen is determined by heuristics.

- Common heuristics: highest-level-first heuristics.
  - The priority of each operation is the length of the longest path in the data dependence graph starting from this operation.

- Code quality: often within 10% from local optimum (inside basic blocks)

Characteristics of DSPs

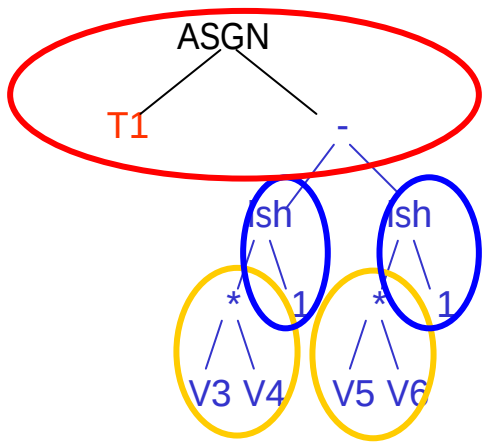
- Cost constraints, low power requirements and specialization:
  - Irregularity
  - Phase coupling problems during code generation
  - Need for specialized algorithms

The Phase Coupling Problem

- Code selection: NP-complete
- Register allocation: NP-complete
- Instruction scheduling: NP-complete
- All of them are interdependent, i.e. decisions made in one phase may impose restrictions to the other.
- Usually they are addressed by heuristic methods in separate phases.
- Thus: often suboptimal combination of suboptimal partial results.
- Moreover: specific/irregular hardware features not well covered by standard code generation methods.

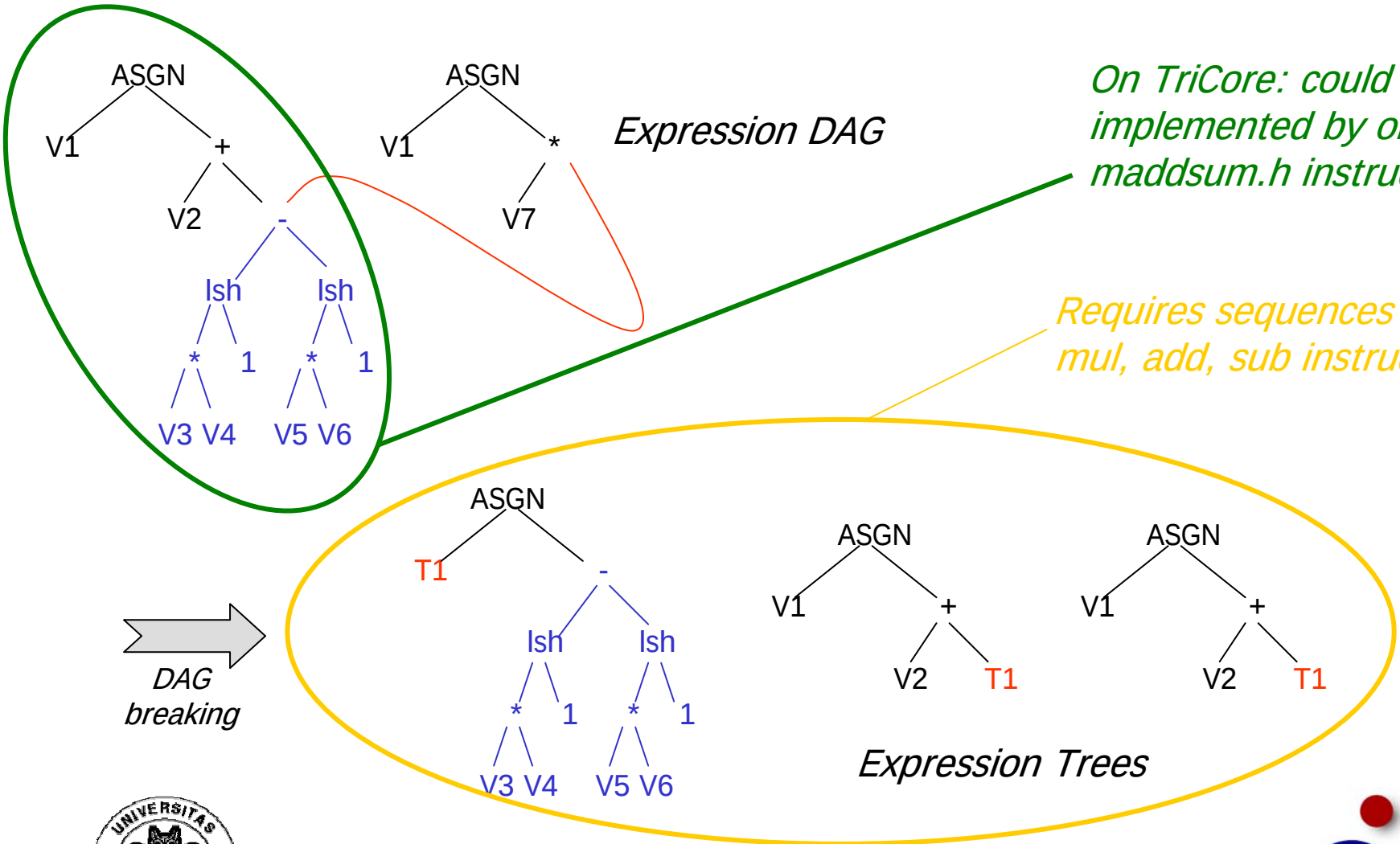
Code Selection and CISC Instructions

- Code selection usually is done by tree parsing (tree pattern matching) and dynamic programming.
- Usually the IR however takes the form of directed acyclic graphs, e.g. due to common subexpressions.
- Before code selection proper DAGs must be broken into trees for the code selection algorithm to work.
- Breaking the trees is done heuristically. Thus the resulting trees and the resulting use of temporary storage locations may destroy the opportunity of generating complex instructions which would correspond to larger expression trees.



mul
mul
shift
shift
sub

Code Selection and CISC Instructions

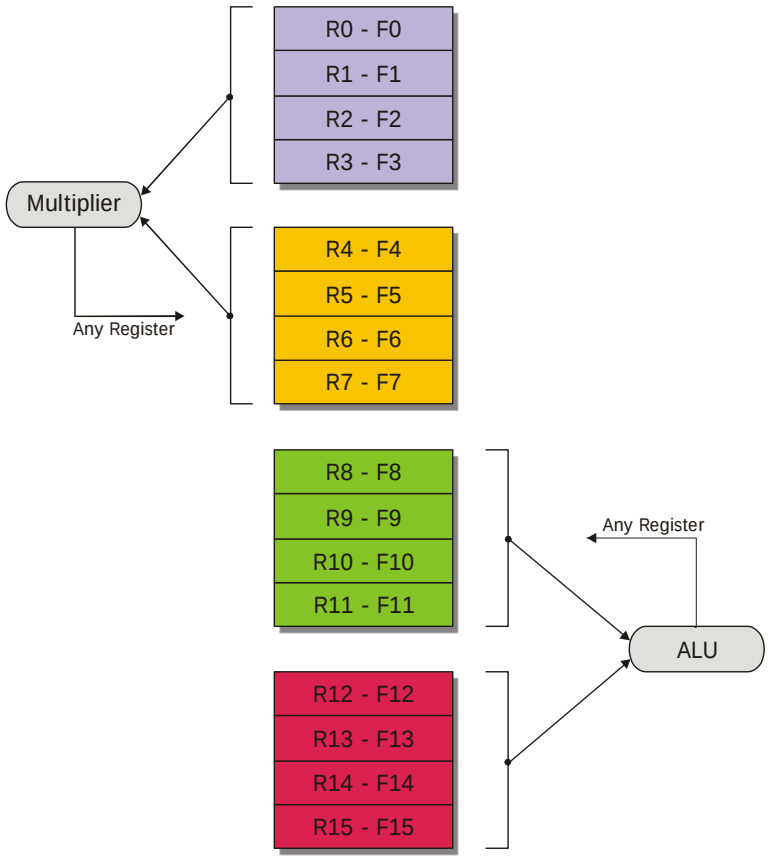


Phase Coupling: Register Allocation and Instruction Scheduling

- Goal of register allocation: minimize number of registers.
  - Consequence: false dependences caused by register reuse, limiting instruction-level parallelism.
- Goal of instruction scheduling: maximize instruction-level parallelism.
  - Consequence: parallelization often forbits register reuse, possibly triggering generation of spill code during register allocation.
- Whichever task is executed first: it can make decisions which are globally suboptimal due to restrictions they impose to the second task.

Phase Coupling Problem: Register Assignment and Scheduling (Analog Devices SHARC)

- Restricted parallelism between ALU and multiplier.



R1=R1*R4
R2=R8+R12

R1 = R1 ∗ R4, R2 = R8 + R12

R1 = R1 ∗ R3
R2 = R8 + R12

R1 = R1 ∗ R3

R2 = R8 + R12

Advanced Topics

- Special compilation goals:
  - usually average-case execution time is optimized
  - compilation for size
  - compilation for energy
  - compilation for worst-case execution time
- Retargetability:
  - Machine description languages for automatically generating code generators.
  - Goal: Keep porting effort low by combining generic with generative mechanisms.
- Integration of code generation phases by exact methods
  - Integer programming
  - Constrained logic programming
  - Branch & Bound algorithms